МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

Тема: Максимальный поток

Студент гр. 0304 _____ Алексеев Р.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2022

Цель работы.

Изучить способы нахождения максимального потока в сети.

Реализовать алгоритм Форда-Фалкерсона.

Задание.

Вариант 2

Поиск в ширину. Обработка совокупности вершин текущего фронта как единого целого, дуги выбираются в порядке уменьшения остаточных пропускных способностей.

Найти максимальный поток в сети, а также фактическую величину потока, протекающего через каждое ребро, используя алгоритм Форда-Фалкерсона.

Сеть (ориентированный взвешенный граф) представляется в виде триплета из имён вершин и целого неотрицательного числа - пропускной способности (веса).

Входные данные:

N - количество ориентированных рёбер графа

v_0 - исток

v_n - сток

v_i v_j ω_{ij} - ребро графа

v_i v_j ω_{ij} - ребро графа

...

Выходные данные:

P_{max} - величина максимального потока

v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

v_i v_j ω_{ij} - ребро графа с фактической величиной протекающего потока

...

В ответе выходные рёбра отсортируйте в лексикографическом порядке по первой вершине, потом по второй (в ответе должны присутствовать все указанные входные рёбра, даже если поток в них равен 0).

Sample Input:

```
7
a
f
a b 7
a c 6
b d 6
c f 9
d e 3
d f 4
e c 2
```

Sample Output:

```
12
a b 6
a c 6
b d 6
c f 8
d e 2
d f 4
e c 2
```

Выполнение работы.

Описание алгоритма

Для нахождения максимального потока в сети был использован алгоритм Форда-Фалкерсона в ширину.

Шаги алгоритма Форда-Фалкерсона:

1. При помощи поиска в ширину находится кратчайший путь от истока до стока.
2. Остаточная пропускная способность ребер, входящих в путь, уменьшается на минимальную остаточную пропускную способность ребра в пути. Аналогично увеличивается остаточная пропускная способность противоположных ребер.
3. Если путь построить не удалось, алгоритм завершает работу.
4. Для поиска пути от истока к стоку был использован поиск в ширину.

Шаги поиска в ширину:

1. Из текущего фронта выбирается вершина.
2. Если дети текущей вершины не были ранее добавлены в предыдущие фронты, то они добавляются в новый фронт.
3. Если дети были добавлены в новый фронт ранее, то сохраняется тот родитель, который смежен ребру с большей остаточной пропускной способностью.
4. Шаги 1-3 повторяются пока не будут обработаны все вершины графа.

Для определения максимального потока через сеть сравниваются пропускные способности ребер, ведущих в сток, до и после алгоритма. Сумма разностей до и после равняется максимальному потоку.

Сложность алгоритма

Каждый шаг алгоритма увеличивает значение потока в сети минимум на 1, следовательно алгоритм завершит свою работу не более чем за f шагов, где f — максимальный поток в сети, т. е. сложность по числу операций - O(f). На каждом шаге алгоритма необходимо найти путь от истока к стоку, т. к. для этого используется поиск в ширину, то сложность поиска по числу операций — O(E), где E - число ребер в сети. Следовательно сложность по числу операций - O(Ef).

Для хранения графа используется список смежности, следовательно сложность по памяти — O(E). Т.к. для реализации алгоритма Форда-Фалкерсона используется список родительских вершин, то сложность алгоритма по памяти O(E) + O(E) = O(E). Для избежания циклов и реализации поиска в ширину, обработанные вершины графа храняться в фронтах, следовательно сложность по памяти — O(V), где V — число вершин графа. Следовательно итоговая сложность по памяти — O($E+V$).

Функции и структуры данных

Для хранеия информации о сети был создан класс *Graf*.

Класс имеет семь полей: *self.start* — истока, *self.finish* — сток, *self.result* — результаты, *self.fronts* — фронты, *self.graf* —словарь для хранения графа в формате пар вершина графа: список исходящих ребер, *self.grafPar* — словарь родительских вершин в формате пар вершина графа: родительская вершина и пропускная способность ребра, по которому перешли, *self.interConc* — флаг промежуточных выводов.

Для работы с вершинами реализован ряд методов.

Метод *createGraf(self)* используется для создания графа из набора ребер. Метод сохраняет родительские и дочерние вершины, а также пропускные способности ребер в поле *self.graf*.

Метод *algFF(self)* используется для реализации алгоритма Форда-Фалкерсона. Метод поочередно запускает поиски в ширину и изменение остаточных пропускных способностей.

Метод *breadthFirstSearch(self)* используется для реализации поиска в ширину. Метод проходит по всем вершинам графа и находит кратчайший путь от истока к стоку с наибольшей остаточной пропускной способностью.

Метод *reduceBandwidth(self)* используется для изменения остаточной пропускной способности ребер. Метод проходит от стока к истоку, определяет минимальную остаточную пропускную способность ребра в пути и уменьшает на её величину остаточные пропускные способности всех ребер пути, аналогично увеличивая противоположные ребра. Если путь построить удалось, то метод возвращает *False*, иначе *True* и вызывает метод *createResult(self)*.

Метод *createResult(self)* используется для формирования результата. Метод перебирает все ребра графа, те, которые были в изначальном графе, сохраняются в результат. После добавление всех ребер результат сортируется лексикографически. Для нахождения максимального потока метод складывает разности между остаточными пропускными способностями до и после работы алгоритма у ребер, ведущих в сток.

Метод *printResult(self)* используется для вывода результат. Метод построчно выводит результаты на экран.

Выводы.

Был изучен алгоритм Форда-Фалкерсона для нахождения максимального потока в сети.

Была написанна программа, находящая максимальный поток в сети при помощи алгоритма Форда-Фалкерсона и поиска в ширину.

Было установлено, что сложность алгоримта Форда-Фалкерсона по числу операций — O(Ef), где E — число ребер в графе, f — максимальный поток, сложность по памяти — O($E+V$), где V — число вершин графа.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
class Graf:     # Граф
    def __init__(self, start, finish, edges, interConc):
        self.start = start              # Исток
        self.finish = finish            # Сток
        self.result = []                # Результаты
         self.fronts = [{self.start: 0}] # Текущий набор фронтов для поиска в ширину
        self.graf = {}                  # Граф
         self.grafPar = {}               # Словарь вида: вершина: родитель, пропуск. способность.
        self.interConc = interConc      # Промежуточные выводы
        self.createGraf(edges)          # Создание графа

    def createGraf(self, edges):    # Создание графа
        if self.interConc:
            print("Создание графа:")
        for edge in edges:  # Перебор всех ребер
            if self.interConc:
                                        print(f"\tДобавление  в  граф ребра \"{edge[0]}\" \"{edge[1]}\" {edge[2]}")

            fV = edge[0]    # Вершина, откуда исходит ребро
            sV = edge[1]    # Вершина, в которую входит ребро

              if self.graf.get(fV):   # Если родительская вершина уже есть в графе
                  if self.interConc:
                            print(f"\t\tВершина \"{fV}\" уже есть в графе. Добавляем новое исходящее из неё ребро")
                   self.graf[fV][sV] = [int(edge[2]), int(edge[2]), True]
              else:   # Если родительской вершины ещё нет в графе
                  if self.interConc:
                         print(f"\t\tВершины \"{fV}\" нет в графе, добаляем её. Добавляем исходящее из неё ребро")
                  self.graf[fV] = {sV: [int(edge[2]), int(edge[2]), True]}

               if self.graf.get(sV):   # Если дочерняя вершина есть в графе
                  if self.interConc:
                            print(f"\t\tВершина \"{sV}\" уже есть в графе. Добавляем новое входящее в неё ребро")
                     if not self.graf[sV].get(fV): # Если ребра до родительской вершины ещё нет
                         self.graf[sV][fV] = [0, 0, False]
              else:   # Если дочерней вершины ещё нет в графе
                  if self.interConc:
```

```
                            print(f"\t\tВершины \"{sV}\" нет в графе,
добаляем её. Добавляем входящее в неё ребро")
                        self.graf[sV] = {fV: [0, 0, False]}
                    if self.interConc:
                        print()

            def algFF(self):    # Алгоритм Форда-Фалкерсона
                if self.interConc:
                    print("Запуск алгоритма Фодра-Фалкерсона:")

                while True:     # Повторяем шаги, пока не закончатся пути
                    self.grafPar = {self.start: [None, 0]}

                    self.breadthFirstSearch()   # Поиск в ширину

                    self.fronts = [{self.start: 0}]

                      if self.reduceBandwidth():  # Уменьшение пропускных
способностей
                        break

            def breadthFirstSearch(self):   # Поиск в ширину
                if self.interConc:
                    print("\tЗапуск поиска в ширину:")
                for front in self.fronts:   # Проходим по всем фронтам
                    if self.interConc:
                        print(f"\t\tТекущий фронт: ", end='')
                        for f in front:
                            print(f"{f} ", end='')
                        print()
                        for ver in front:         # Проходи по вершинам
текущего фронта
                        if self.interConc:
                            print(f"\t\tОбрабатываем вершину \"{ver}\":")
                           for newVer in self.graf[ver]:    # Проходим по
дочерним вершинам текущей вершины
                             if self.interConc:
                                 print(f"\t\t\tПотомок \"{newVer}\":")
                              ed = self.grafPar.get(newVer)   # Проверка,
попадали ли ранее в дочернюю вершину

                             if front == self.fronts[-1]:    # СОздание
нового фронта по необходимости
                              self.fronts.append({})

                             if ed:  # Если в дочернюю вершину уже есть
путь
                              if self.interConc:
                                  print(f"\t\t\t\tУже есть путь")
                                 i = self.fronts[-1].get(newVer)     #
Лежит ли дочерняя вершина в текущем фронте

                              # Если дочерняя вершина в текущем фронте,
старая пропускная способность меньше новой и
```

```
                                # до дочерней вершины можно пройти по
ребру
                                    if i and self.fronts[-1][newVer] <
self.graf[ver][newVer][0] and self.graf[ver][newVer][0] > 0:
                                if self.interConc:
                                    print(f"\t\t\t\tЗаменяем путь на
новый, т.к. пропуск. способность нового больше: "
                                          f"{self.graf[ver][newVer]
[0]}")
                                self.grafPar[newVer] = [ver,
self.graf[ver][newVer][0]] # Заменяем родителя на нового
                                self.fronts[-1][newVer] =
self.graf[ver][newVer][0]     # Заменяем пропуск. спос. на новую

                            elif self.graf[ver][newVer][0] > 0: # Если в
дочернюю вершину ещё не заходили
                                if self.interConc:
                                    print(f"\t\t\t\tПути ещё нет")
                                    print(f"\t\t\t\tДобавляем путь с
пропуск. способностью: {self.graf[ver][newVer][0]}")
                                self.fronts[-1][newVer] = self.graf[ver]
[newVer][0]     # Добавляем дочернюю вершину в фронт
                                self.grafPar[newVer] = [ver,
self.graf[ver][newVer][0]] # Добавляем родителя
                    if self.interConc:
                        print()

            def reduceBandwidth(self):    # Уменьшение пропускных
способностей
            if self.interConc:
                print(f"\tЗапуск уменьшения пропуск. способности:")
                ver = self.grafPar.get(self.finish) # В качестве
начальной вершины - сток

            if ver == None:       # Если из стока никуда нельзя
попасть, значит путей до стока нет
                if self.interConc:
                    print(f"\t\tПути из истока до стока нет\n")
                self.createResult()
                return True

            minBW = -1  # Минимальный вес в пути

            while True:     # Проходим по вершинам, пока не дойдем до
истока
                # Если минимальная пропуск. способность большей
текущей
                if (minBW > ver[1] or minBW == -1) and ver[0] !=
None:
                    minBW = ver[1]

                ver = self.grafPar.get(ver[0])
                if ver == None: # Если дошли до истока
                    break
            if self.interConc:
```

```
                print(f"\t\tМинимальная пропуск. способность пути: {minBW}")

            sV = self.finish    # Вершина, в которую входит ребро
            fV = self.grafPar[self.finish][0]   # Вершина, из которой исходит ребро
            while True:
                self.graf[fV][sV][0] -= minBW   # Уменьшаем пропуск. способность ребра
                    self.graf[sV][fV][0] += minBW    # Увеличиваем пропуск. способность противополож. ребра
                sV = fV
                fV = self.grafPar[fV][0]
                if sV == self.start or fV == None:  # Если достигли истока
                    break
            if self.interConc:
                print()
            return False

        def createResult(self):     # Формирование результата
            if self.interConc:
                print(f"\tЗапуск формирования результата:")
            flow = 0    # Максимальный поток
            edges = []  # Ребра графа
            for ver in self.graf:   # Перебираем все вершины графа
                    for newVer in self.graf[ver]:    # Перебираем все дочерние вершины
                        if self.graf[ver][newVer][2]:   # Если ребро в дочернюю вершину было в изначальном графе
                                edges.append([ver, newVer, self.graf[ver][newVer][1] - self.graf[ver][newVer][0]])
                            if self.interConc:
                                                print(f"\t\tДобавляем ребро    \"{ver}\"    \"{newVer}\"    {self.graf[ver][newVer][1]    - self.graf[ver][newVer][0]} в результат")
                            if edges[-1][2] < 0:
                                edges[-1][2] = 0
                                if newVer == self.finish:   # Если ребро входит в сток
                                    if self.interConc:
                                        print(f"\t\tУвеличиваем макс. поток на "
                                              f"{self.graf[ver][newVer][1] - self.graf[ver][newVer][0]}:")
                                    flow += self.graf[ver][newVer][1] - self.graf[ver][newVer][0]

            self.result.append(flow)
            if self.interConc:
                print(f"\t\tТекущий результат:")
                print(f"\t\t\t{flow}")
                for i in range(len(edges)):
                            print(f"\t\t\t{edges[i][0]} {edges[i][1]} {edges[i][2]}")
```

```
                print(f"\t\tСортируем результат\n")
            edges.sort()
            self.result += edges

        def printResult(self):  # Вывод результата
            if self.interConc:
                print("Результат:")
            for i in range(len(self.result)):
                if i == 0:
                    print(self.result[i])
                else:
                      print(f"{self.result[i][0]} {self.result[i][1]}
{self.result[i][2]}")


    if __name__ == "__main__":
        n = int(input())
        start = input()
        finish = input()

        edges = []
        for _ in range(n):
            edges.append(list(input().split()))

        graf = Graf(start, finish, edges, True)
        graf.algFF()
        graf.printResult()
```

ПРИЛОЖЕНИЕ Б

ТЕСТИРОВАНИЕ

Таблица Б.1 - Тестирование

№ п/п	Входные данные	Выходные данные	Комментарии
1.	15 s f s a 15 s b 9 s c 17 a d 9 a e 13 d a 21 b e 14 c h 11 h c 4 e c 7 c f 25 d f 16 e f 8 h f 11 e h 7	41 a d 9 a e 6 b e 9 c f 18 c h 0 d a 0 d f 9 e c 1 e f 8 e h 6 h c 0 h f 6 s a 15 s b 9 s c 17	
2.	7 a f a b 7 a c 6 b d 6 c f 9 d e 3 d f 4 e c 2	12 a b 6 a c 6 b d 6 c f 8 d e 2 d f 4 e c 2	
3.	9	16	

	s f s a 6 s b 3 a c 4 d f 9 b d 12 c f 7 a b 5 b c 11 s d 8	a b 1 a c 4 b c 3 b d 1 c f 7 d f 9 s a 5 s b 3 s d 8	
4.	6 s f s a 6 s b 3 a c 4 d f 9 b d 12 c f 7	7 a c 4 b d 3 c f 4 d f 3 s a 4 s b 3	
5.	14 s f s a 15 s b 9 s c 17 a d 9 a e 13 d a 21 b e 14 c h 12 h c 4 e c 7	36 a d 9 a e 2 b e 8 c f 8 c h 11 d a 0 d f 9 e c 2 e f 8 h c 0 h f 11 s a 11	

	c f 8 d f 16 e f 8 h f 11	s b 8 s c 17	